Logical Methods in Computer Science
Vol. 7 (2:17) 2011, pp. 1–33
www.lmcs-online.org

Submitted Jun. 15, 2010
Published Jun. 21, 2011

AMORTISED RESOURCE ANALYSIS WITH SEPARATION LOGIC

ROBERT ATKEY

University of Strathclyde, UK
e-mail address: Robert.Atkey@cis.strath.ac.uk

Abstract. Type-based amortised resource analysis following Hofmann and Jost—where resources are associated with individual elements of data structures and doled out to the programmer under a linear typing discipline—have been successful in providing concrete resource bounds for functional programs, with good support for inference. In this work we translate the idea of amortised resource analysis to imperative pointer-manipulating languages by embedding a logic of resources, based on the affine intuitionistic Logic of Bunched Implications, within Separation Logic. The Separation Logic component allows us to assert the presence and shape of mutable data structures on the heap, while the resource component allows us to state the consumable resources associated with each member of the structure.

We present the logic on a small imperative language, based on Java bytecode, with procedures and mutable heap. We have formalised the logic and its soundness property within the Coq proof assistant and extracted a certified verification condition generator. We also describe an proof search procedure that allows generated verification conditions to be discharged while using linear programming to infer consumable resource annotations.

We demonstrate the logic on some examples, including proving the termination of in-place list reversal on lists with cyclic tails.

1. Introduction

Tarjan, in his paper introducing the concept of amortised complexity analysis [25], noted that the statement and proof of complexity bounds for operations on some data structures can be simplified if we think of the data structure as being able to store "credits" that are used up by later operations. By setting aside credit inside a data structure to be used by later operations, the cost of a sequence of operations can be amortised over time. In this paper, we propose a way to merge amortised complexity analysis with Separation Logic [19, 24] to formalise some of these arguments and to simplify the specification and verification of resource-consuming pointer-manipulating programs.

Separation Logic is built upon a notion of resources and their separation. The assertion $A * B$ holds for some resource if it can be split into two separate resources that make A true and B true respectively. Resource separation enables local reasoning about mutation of resources; if the program mutates the resource associated with A, then we know that B

1998 ACM Subject Classification: D.2.4, F.3.1.

Key words and phrases: resource bounded computing, amortised analysis, separation logic, java bytecode, program logic, resource models.

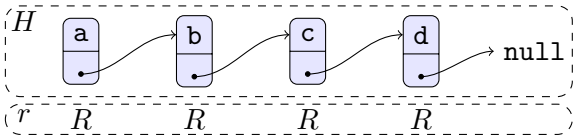
DOI:10.2168/LMCS-7 (2:17) 2011

is still true on its separate resource. Usually Separation Logic uses mutable heaps as its notion of resource. In this paper, we combine heaps with consumable resources to reason about resource consumption as well as resource mutation.

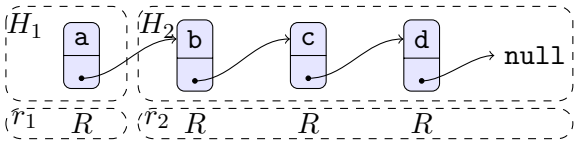
To see how Separation Logic-style reasoning about consumable resources is beneficial, consider the standard inductively defined list segment predicate from Separation Logic, augmented with an additional proposition R denoting the presence of a consumable resource for every element of the list:

$$\begin{aligned}\mathsf{lseg}(R, x, y) \equiv \quad & x = y \wedge \mathsf{emp} \\ & \vee \exists d, z.\ [x \overset{\mathsf{data}}{\mapsto} d] * [x \overset{\mathsf{next}}{\mapsto} z] * R * \mathsf{lseg}(z, y)\end{aligned}$$

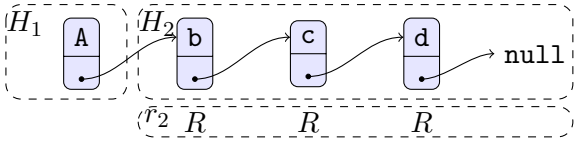
We will introduce the assertion logic properly in Section 4 below. We can represent a heap H and a consumable resource r that satisfy this predicate graphically:



So we have $r, H \models \mathsf{lseg}(R, x, \mathit{null})$, assuming x contains the address of the head of the list. Here $r = R \cdot R \cdot R \cdot R$—we assume that consumable resources form a commutative monoid—and r represents the resource that is available for the program to use in the future. We can split H and r to separate out the head of the list with its associated resource:



This heap and resource satisfy $r_1 \cdot r_2, H_1 \uplus H_2 \models [x \overset{\mathsf{data}}{\mapsto} \mathtt{a}] * [x \overset{\mathsf{next}}{\mapsto} y] * R * \mathsf{lseg}(R, y, \mathit{null})$, where $H_1 \uplus H_2 = H$, $r_1 \cdot r_2 = r$ and we assume that y contains the address of the `b` element. Now that we have separated out the head of the list and its associated consumable resource, we are free to mutate the heap H_1 and consume the resource r_1 without affecting the tail of the list. So the program can move to a new state where the head of the list has been mutated to `A` and the associated resource has been consumed:



We do not need to do anything special to reason that the tail of the list and its associated consumable resource are unaffected.

The combined assertion about heap and consumable resource describes the current shape and contents of the heap and also the available resource that the program may consume in the future. By ensuring that, for every state in the program's execution, the resource consumed plus the resource available for consumption in the future is less than or equal to a predefined bound, we can ensure that the entire execution is resource bounded. This is the main assertion of soundness for our program logic in Section 3.5. We also treat a variant program logic that allows dynamic but fallible resource acquisition in Section 3.6.

By intermixing resource assertions with Separation Logic assertions about the shapes of data structures, as we have done with the resource carrying lseg predicate above, we can

specify amounts of resource that depend on the shape of data structures in memory. By the definition of lseg, we know that the amount of resource available to the program is linearly proportional to the length of the list, without having to do any arithmetic reasoning about lengths of lists. In Section 4.2 we describe some useful inductively defined predicates that maintain a close connection between shape and resources.

The association of resources with parts of a data structure is exactly the banker's approach to amortised complexity analysis proposed by Tarjan.

Our original inspiration for this work came from the work of Hofmann and Jost [15] on the automatic heap-space analysis of functional programs. Their analysis associates with every element of a data structure a permission to use a piece of resource (in their case, heap space). This resource is made available to the program when the data structure is decomposed using pattern matching. When constructing part of a data structure, the required resources must be available. A linear type system is used to ensure that data structures carrying resources are not duplicated since this would entail duplication of consumable resource. This scheme was later extended to imperative object-oriented languages [16, 17], but still using a type-based analysis.

1.1. **Contributions.** We summarise the content and contributions of this work:

- In Section 3, we define a program logic that allows mixing of assertions about heap shapes and assertions about future consumable resources. Tying these together allows us to easily state resource properties in terms of the shapes of heap-based data structures, rather than in terms of extensional properties such as their size or contents. We have formalised the soundness proof of our program logic in the Coq proof assistant.
- In Section 4 we present a syntax for our assertion logic, based on a combination of Boolean Bunched Implications extended with pointer assertion primitives, as is usual for Separation Logic, and affine intuitionistic Bunched Implications to declaratively state properties of the consumable resource available to a program. We also discuss several useful inductively defined predicates for this logic that demonstrate tight connections between heap shapes and resources. We also discuss the advantages and disadvantages of this tight connection.
- In Section 5, we define a restricted subset of the assertion logic that allows us to perform effective proof search to discharge verification conditions, while inferring resource annotations. A particular feature of the proof search procedure we describe is that, given loop invariants that specify only the the shape of data structures, we can infer the necessary consumable resource annotations.
- In Section 2 and Section 6, we demonstrate the logic on some examples, showing how a mixture of amortised resource analysis and Separation Logic can be used to simplify resource-aware specifications, deal with relatively complex pointer manipulation and to prove termination in the presence of cyclic structures in the heap.

1.2. **Differences to previously published versions.** This paper is a revised and expanded version of the ESOP 2010 conference version [5]. Additional explanation has been provided and Section 3.6 has been added on a variant program logic accounting for dynamic resource acquisition. Section 4 has been expanded with more examples of inductively defined predicates tightly integrating shape and resource properties, and a discussion of the disadvantages of this integration. Section 5 on automated verification and proof search has

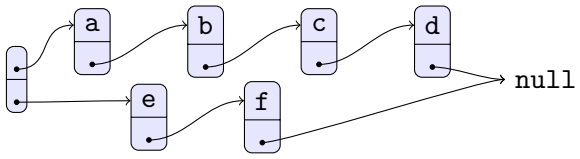
been refined for clarity and to better match the implementation. We have also included the merge-sort example in Section 2.2 from an invited TGC 2010 paper [2].

2. Motivating Examples

The example we gave in the introduction, where a program iterates through a list consuming resources as it proceeds, only demonstrates an extremely simple, albeit common, pattern. In this section, we give two more complex examples that serve to highlight the advantages of the amortised approach to specifying and verifying resource bounds. The first example, in Section 2.1, demonstrates how the integrated description of available consumable resources and heap structure helps with specification. The second example, in Section 2.2, shows how the amortised approach simplifies the problem of verifying the resource consumption of a pointer manipulating program. This example also shows a weakness of maintaining a tight connection between heap shape and consumable resources, which we discuss further in Section 4.3.

In each case we attempt to demonstrate how amortised reasoning is easier than the traditional approach of keeping a global counter for consumed resources as a "ghost" variable in the logic.

2.1. **Functional Queues.** We consider so-called functional queues [18, 8], where a queue is represented by a pair of lists. This example is a standard one for introducing amortised complexity analysis [21]. We verify an imperative implementation that performs mutations in-place on the underlying lists. The point of this example is to see how the amortised technique simplifies the specifications of the procedures operating on this data structure.



The top list represents the head of the queue, while the bottom list represents the tail of the queue in reverse. This structure represents the queue $[\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{f}, \mathtt{e}]$. When we enqueue a new element, we add it to the head of the bottom list. To dequeue an element, we remove it from the head of the top list. If the top list is empty, then we reverse the bottom list and change the top pointer to point to it, changing the bottom pointer to point to *null*, representing the empty list.

When determining the complexity of these operations, it is obvious that the enqueue operation is constant time, but the dequeue operation either takes constant time if the top list is empty, or takes time linear in the size of the bottom list in order to perform the reversal. If we were to account for resource usage by maintaining a global counter then we would have to expose the lengths of the two lists in specification of the enqueue and dequeue instructions. So we would need a predicate $\mathsf{queue}(x, h, t)$ to state that x points to a queue with a head and tail lists of lengths h and t respectively. The operations would have the specifications (written as Hoare triples):

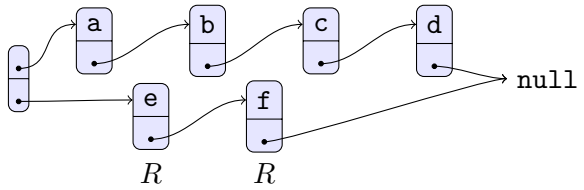
$$\begin{array}{lrcl}
\forall r_1. & \{\mathsf{queue}(x,h,t) \wedge r_c = r_1\} & \mathtt{enqueue} & \{\mathsf{queue}(x,h,t+1) \wedge r_c = r_1 + R\} \\
\forall r_1. & \{\mathsf{queue}(x,0,t) \wedge r_c = r_1\} & \mathtt{dequeue} & \{\mathsf{queue}(x,t \dot{-} 1,0) \wedge r_c = r_1 + (1+t)R\} \\
\forall r_1. & \{\mathsf{queue}(x,h+1,t) \wedge r_c = r_1\} & \mathtt{dequeue} & \{\mathsf{queue}(x,h,t) \wedge r_c = r_1 + R\}
\end{array}$$

where r_c is a ghost variable counting the total amount of resource consumed by the program, and R is the amount of resource required to perform a single list node manipulation. The dotted minus $t \dot{-} 1$ denotes the predecessor operation on natural numbers. Note that we have had to give two specifications for `dequeue` for the cases when the head list is empty and when the head list has an element. The accounting for the sizes of the internals of the queue data structure is of no interest to clients of this data structure. These specifications complicate clients' reasoning when using these queues.

Using amortised analysis, this specification can be drastically simplified. We associate a single piece of resource with each element of the tail list so that when we come to reverse the list we have the necessary resource available to reverse each list element. The queue predicate is therefore:

$$\mathsf{queue}(x) \equiv \exists y, z.\ [x \overset{\mathsf{head}}{\mapsto} y] * [x \overset{\mathsf{tail}}{\mapsto} z] * \mathsf{lseg}(e, y, null) * \mathsf{lseg}(R, z, null)$$

where lseg is the resource-carrying list predicate given above and e is the unit of the consumable resource monoid, representing no resource. The list holding the head of the queue has no resource associated with every element, while the list holding the tail of the queue does have a resource associated with every element, waiting to pay for the list reversal when it occurs. Diagrammatically, a queue with associated resources looks like this:



The specifications of the operations now becomes straightforward:

$$\{\mathsf{queue}(x) * R * R\}\texttt{enqueue}\{\mathsf{queue}(x)\} \qquad \{\mathsf{queue}(x) * R\}\texttt{dequeue}\{\mathsf{queue}(x)\}$$

To enqueue an element, we require two elements of resource: one to add the new element to the tail list, and one to "store" in the list so that we may use it for a future reversal operation. To dequeue an element, we require a single element of resource to remove an element from a list. If a list reversal is required then it is paid for by the resources previously required by `enqueue`.

Once we have set the specification of queues to store one element of resource for every node in the tail list, we can use the resource annotation inference procedure presented in Section 5 to generate the resource parts of the `enqueue` and `dequeue` specifications.

2.2. **Merge-sort Inner Loop.** We now describe a more complicated list manipulating program that shows the benefits of the amortised approach for verification. This example demonstrates the combination of reasonably complex pointer manipulation with resource reasoning. Most of the technical details arise from dealing with the heap-shape behaviour of the program; the resource bounds simply drop out thanks to the inference of resource annotations.

Consider the Java method declaration `mergeInner` shown in Figure 1 that describes the inner loop of an in-place merge sort algorithm for linked lists[1]. The method takes two arguments: `list`, a reference to the head node of a linked list; and `k`, an integer. The integer argument dictates the sizes of the sublists that the method will be merging in this

[1]Adapted from C code: `http://www.chiark.greenend.org.uk/~sgtatham/algorithms/listsort.html`.

```
public static Node mergeInner (Node list, int k) {
     Node p      = list;
     Node tail   = null;

     list = null;

     while (p != null) {
          Node q = p;
          for (int i = 0; i < k; i++) {
               q = q.next;
               if (q == null) break;
          }

          Node pstop = q;
          int qsize = k;
          while (p != pstop || (qsize > 0 && q != null)) {
               Node e;
               if (p == pstop) {
                    e = q;
                    q = q.next;
                    qsize--;
               } else if (qsize == 0 || q == null) {
                    e = p;
                    p = p.next;
               } else if (p.data <= q.data) {
                    e = p;
                    p = p.next;
               } else {
                    e = q;        // perform swap
                    q = q.next;
                    qsize--;
               }

               if (tail != null)  tail.next = e;
               else               list = e;

               tail = e;
          }

          p = q;
     }

     if (tail == null) return null; else tail.next = null;
     return list;
}
```

Figure 1: Inner loop of an in-place linked-list merge sort

pass. The method steps through the list `2*k` elements at a time, merging the two sublists of length `k` each time. The outer loop does the `2*k` stepping, and the inner loop does the merging. To accomplish a full merge sort, `mergeInner` would be called $\log_2 n$ times with doubling `k`, where n is the length of the list.

Assume that we wish to account for the number of swapping operations performed by this method, i.e. the number of times that the fourth branch of the `if` statement in the inner loop is executed. We accomplish this in our implementation by inserting a special `consume` instruction at this point.

The pre- and post-conditions of the method are as follows:

$$\begin{aligned} \text{Pre}(\texttt{mergeInner}) &: \texttt{list} \neq \mathit{null} \wedge (\mathsf{lseg}(x, \texttt{list}, \mathit{null}) * R^y) \\ \text{Post}(\texttt{mergeInner}) &: \mathsf{lseg}(0, \texttt{retval}, \mathit{null}) \end{aligned}$$

The precondition states that the first argument points to a list segment ending with `null`, with x amount of resource associated with every element of the list, and y amount of additional resource that may be used. The actual values of x and y will be inferred by a linear program solver. The condition $\texttt{list} \neq \mathit{null}$ is a safety condition required to prevent a null pointer exception.

The outer loop in the method needs a disjunctive invariant corresponding to whether this is the first iteration or a later iteration.

$$\begin{array}{ll} & (\mathsf{lseg}(o_1, \texttt{list}, \texttt{tail}) * [\texttt{tail} \overset{\mathsf{next}}{\mapsto} ?] * [\texttt{tail} \overset{\mathsf{data}}{\mapsto} ?] * \mathsf{lseg}(o_2, \texttt{p}, \mathit{null}) * R^{o_3}) \\ \vee & ((\texttt{list} = \mathit{null} \wedge \texttt{tail} = \mathit{null}) * \mathsf{lseg}(o_4, \texttt{p}, \mathit{null}) * R^{o_5}) \end{array}$$

The first disjunct is used on normal iterations of loop: the variable `list` points to the list that has been processed so far, ending at `tail`; and `p` points to the remainder of the list that is to be processed. We have annotated these lists with the resource variables o_1 and o_2 that will contain the resources associated with each element of these lists. The second disjunct covers the case of the first iteration, when `list` and `tail` are null and `p` points to the complete list to be processed. The resource annotation o_4 will be filled in with the amount of resource that is required for every element of the list to be processed. As one might expect, this will be equal to the value of x, the amount of resource required by the precondition of the method.

Moving on, we consider the first inner loop that advances the pointer `q` by `k` elements forward, thus splitting the list ahead of `p` into a `k`-element segment and the rest of the list. The next loop will merge the first `k`-element segment with the `k`-element prefix of the second segment. It is convenient for our implementation to split out this inner loop into another method[2], with the following signature:

```
public static Node advance (Node l, int k)
```

The argument `l` points to a linked list, and the method will advance `k` elements through the list (or until the end) and return a pointer to the split point. The specification of this method is:

$$\begin{aligned} \text{Pre}(\texttt{advance}) &: \mathsf{lseg}(a_0, \texttt{l}, \mathit{null}) \\ \text{Post}(\texttt{advance}) &: \mathsf{lseg}(a_0, \texttt{l}, \texttt{retval}) * \mathsf{lseg}(a_0, \texttt{retval}, \mathit{null}) \end{aligned}$$

[2]This is because our implementation works on unstructured JVM-like bytecode, and so cannot easily apply Separation Logic's frame rule to modularly reason about the loop. Using a separate method allows application of the frame rule. See Remark 3.2.

Again, we have left the resource annotation on the elements of the list as a variable a_0, to be filled in by the linear solver. The appearance of the same variable in the pre- and post-condition implies that we expect this resource to be preserved by the method. The fact that we have had to explicitly mention the resources that must be preserved points to a limitation of the amortised method as we present it here. We discuss this issue further in Section 4.3.

Proceeding though our main method, the invariant of the inner loop is as follows, again as two disjuncts according to whether it is the first or later iteration of the outer loop:

$$\begin{array}{rl} & (\mathsf{lseg}(i_1, \mathtt{list}, \mathtt{tail}) * [\mathtt{tail} \overset{\mathsf{next}}{\mapsto} ?] * [\mathtt{tail} \overset{\mathsf{data}}{\mapsto} ?] \\ & \qquad * \mathsf{lseg}(i_2, \mathtt{p}, \mathtt{pstop}) * \mathsf{lseg}(i_3, \mathtt{q}, \mathit{null}) * R^{i_4}) \\ \vee & ((\mathtt{list} = \mathit{null} \wedge \mathtt{tail} = \mathit{null}) * \mathsf{lseg}(i_5, \mathtt{p}, \mathtt{pstop}) * \mathsf{lseg}(i_6, \mathtt{q}, \mathit{null}) * R^{i_7}) \end{array}$$

The first part of each disjunct is as before, stating that `list` to `tail` contains the part of list that has been processed. Since we have now split the remainder of the list into two pieces we have two separate list segments referenced by `p` and `q` pointing to the parts of the list that are to be merged. The resource meta-variable i_1 will indicate the amount of resource associated with the list that has been processed; i_2 and i_5 are the amount of resource associated with the "left-hand" pending list; and i_3 and i_4 are the resource associated with the "right-hand" pending list.

At the end of the method, we null terminate the list and return. A superfluous null check on `this` is required for our tool to prove memory safety. The fact that `this` is non-null relies on the execution of the inner loop at least once, which requires that $k > 0$. This fact is not expressible in the logic of the implementation described in Section 5.

Running this example through our implementation produces the solution $x = 1$, $y = 0$ for the precondition's resource annotations. This indicates that the input list needs to contain one element of resource for every list element. For the outer loop's invariant, we obtain $o_2 = o_4 = 1$ and all the others are 0. This indicates that the list we have processed has had all its resources consumed, while the list remaining to be processed still has associated resources. This is as expected for a loop iterating through a list. The specification of `advance` is completed by inferring $a_0 = 1$, indicating that `advance` preserves the resources associated with the list. Finally the inner loop's invariant has $i_2 = i_3 = i_5 = i_6 = 1$ and all others 0, indicating that the two list segments that are remaining to be processed have associated resources, while the processed segments do not.

2.2.1. *Comparisons to other techniques.* Though we have had to work to supply the loop invariants for our implementation, we note that these invariants may be inferred by other tools, for example [6], and the resource variables automatically inserted using the procedure outlined in Section 5. The key to the amortised approach is the tight connection between shape invariants, which is a complex but well-studied problem, and resource consumption.

Most other techniques for resource usage analysis that handle data structures do so by considering the sizes of the structures. The SPEED system of Gulwani et al. [13] can infer resource bounds for programs manipulating heap-based data structures, but only when these data structures are manipulated through abstract interfaces. The specifications for these abstract interfaces record the effect of the operations on the size of the data structure. Thus, the technique is unable to cope with the kind of program that we have presented above that uses direct pointer manipulation. Nevertheless, Gulwani et al. report impressive results on real-world Microsoft product code.

The COSTA system [1] can deal with some uses of direct pointer manipulation, but accounts for the sizes of heap-based data structures by counting the length of the longest path from a given reference. Thus, it cannot deal with programs that demonstrate sharing on the heap; the Java method described above has, in its inner loop, three pointers all pointing to the same list.

One might also use Separation Logic to deal with sharing on the heap, and add information on the sizes of heap-based data structures to account for resource usage. So one would have a predicate $\mathsf{lseg}^n(x, y)$ that describes a list segment of length n from x to y, along with a ghost variable to track resource consumption as discussed in Section 2.1. We argue that the amortised approach described here is simpler due to the differences in reasoning between the *global* property of the length of a whole list, and the *local* property of each list element having an associated amount of resource to be used. For example, consider the specification of the `advance` method using sized structures:

$$\begin{aligned} \mathrm{Pre}(\texttt{advance}) &: \mathsf{lseg}^n(\texttt{l}, \mathit{null}) \\ \mathrm{Post}(\texttt{advance}) &: \exists n_1, n_2.\ n_1 + n_2 = n \wedge (\mathsf{lseg}^{n_1}(\texttt{l}, \texttt{retval}) * \mathsf{lseg}^{n_2}(\texttt{retval}, \mathit{null})) \end{aligned}$$

We have had to introduce two existential variables indicating the sizes of the lists returned by the method. These additional values have to then be related back to the length of the original list by the calling method, and thence to the resource consumption, requiring non-straightforward arithmetic reasoning. The amortised approach exploits the shape-reasoning already present in Separation Logic to account for resources.

3. A Program Logic for Heap and Resources

We now describe a simple programming language and a consumable-resource-aware logic for it. We define a "shallow" program logic where we treat pre- and post-conditions and program assertions as arbitrary predicates over heaps and consumable resources. In Section 4, we will layer on top of this a "deep" assertion logic where predicates are actually Separation Logic formulae augmented with atomic resource propositions.

At the end of this section, we consider a variant system that allows dynamic but fallible resource acquisition as well as resource consumption.

The development in this section has been formalised within the Coq proof assistant; thus the shallow embedding makes use of Coq's own logic as the assertion logic. We also make minor use of Coq's dependent types. Lemma 3.1 and Theorem 3.3 establishing the soundness of the program logic have been mechanically verified within Coq, as have Lemma 3.5 and Theorem 3.6 establishing soundness for the dynamic resource acquisition variant.

3.1. Semantic Domains.

Assume an infinite set $\mathbb{A}$ of memory addresses. We model heaps as finite partial maps $\mathbb{H} = (\mathbb{A} \times \mathbb{F}) \rightharpoonup_{\mathit{fin}} \mathbb{V}$, where $\mathbb{F}$ ranges over field names and $\mathbb{V} = \mathbb{A}_\perp + \mathbb{Z}$ represents the values that programs can directly manipulate: possibly null addresses and integers. We write $\mathit{dom}(H)$ for the domain of a heap and $H_1 \# H_2$ for heaps with disjoint domains; $H_1 \uplus H_2$ denotes union of heaps with disjoint domains.

Consumable resources are represented as elements of an ordered monoid $(\mathcal{R}, \sqsubseteq, \cdot, e)$, where e is the least element. Example consumable resources include $(\mathbb{N}, \leqslant, +, 0)$ or $(\mathbb{Q}^{\geqslant 0}, \leqslant, +, 0)$ for representing a single resource that is consumed (e.g. time or space), or multisets for representing multiple named resources that may be consumed independently. The ordering on consumable resources is used to allow weakening in our assertion logic: we allow

the asserter to assert that more resources are required by the program than are actually needed.

As is standard in the semantics of substructural logics [23], we make use of a ternary relation to describe the combination of separate entities. In our case, the entities are pairs of heaps and consumable resources:

$$Rxyz \;\Leftrightarrow\; H_1 \# H_2 \wedge H_1 \uplus H_2 = H_3 \wedge r_1 \cdot r_2 \sqsubseteq r_3 \quad \text{where } x = (H_1, r_1), y = (H_2, r_2), z = (H_3, r_3)$$

We extend the order on resources to pairs of heaps and resources by $(H_1, r_1) \sqsubseteq (H_2, r_2)$ iff $H_1 = H_2$ and $r_1 \sqsubseteq r_2$.

3.2. **A Little Virtual Machine.** The programming language we treat is a simple stack-based virtual machine, similar to Java bytecode. We have removed the class-based object system and virtual methods, but retained mutable heap and procedures. There are two types: int and ref, corresponding to the two kinds of values in $\mathbb{V}$. We assume a set $\mathbb{P} \ni \mathit{pname}$ of procedure names, where a procedure's name also determines its list of argument types and its return type. Programs are organised into a finite set of procedures, indexed by their name and individually consisting of lists of instructions from the following collection:

$$\begin{aligned}\iota \;::=\;& \mathsf{iconst}\ z \mid \mathsf{ibinop}\ \mathit{op} \mid \mathsf{pop} \mid \mathsf{load}\ n \mid \mathsf{store}\ n \mid \mathsf{aconst\_null} \\ \mid\;& \mathsf{binarycmp}\ \mathit{cmp}\ \mathit{offset} \mid \mathsf{unarycmp}\ \mathit{cmp}\ \mathit{offset} \mid \mathsf{ifnull}\ \mathit{offset} \mid \mathsf{goto}\ \mathit{offset} \\ \mid\;& \mathsf{new}\ \mathit{desc} \mid \mathsf{getfield}\ \mathit{fnm} \mid \mathsf{putfield}\ \mathit{fnm} \mid \mathsf{free}\ \mathit{desc} \mid \mathsf{consume}\ r \\ \mid\;& \mathsf{return} \mid \mathsf{call}\ \mathit{pname}\end{aligned}$$

These instructions—apart from $\mathsf{consume}$—are standard, so we only briefly explain them. Inside each activation frame, the virtual machine maintains an operand stack and a collection of local variables, both of which contain values from the semantic domain $\mathbb{V}$. Local variables are indexed by natural numbers. The instructions in the first two lines of the list perform the standard operations with the operand stack, local variables and program counter. The third line includes instructions that allocate, free and manipulate structures stored in the heap. The instruction $\mathsf{consume}\ r$ consumes the resource r. The *desc* argument to new and free describes the structure to be created on the heap by the fields and their types. The fourth line contains the procedure call and return instructions that manipulate the stack of activation frames.

Individual activation frames are tuples $\langle \mathit{code}, S, L, \mathit{pc} \rangle \in \mathsf{Frm}$ consisting of the list of instructions from the procedure being executed, the operand stack and local variables, and the program counter. The first two lines of instructions that we gave above only operate within a single activation frame and have no effect on the heap or consumable resources, so we give their semantics as a small-step relation between frames: $\xrightarrow{\mathsf{frm}} \subseteq \mathsf{Frm} \times \mathsf{Frm}$ defined in Figure 2. The rules make use of semantic interpretations $[\![\mathit{op}]\!]$ and $[\![\mathit{cmp}]\!]$ of binary operations and binary relations on integers.

The third line of instructions contains those that manipulate the heap and consume resources. Their small-step operational semantics is modelled by a relation $\xrightarrow{\mathsf{mut}} \subseteq \mathsf{Frm} \times \mathbb{H} \times \mathsf{Frm} \times \mathbb{H} \times \mathcal{R}$, which relates the before and after activation frames and heaps, and states the consumable resource consumed by this step. The rules defining this relation are given in Figure 3. The rules for the instructions new and free take a parameter *desc*. This parameter describes the fields and their types for the object to be allocated or deallocated.

For a description $desc = \langle fnm_1 : \tau_1, ..., fnm_n : \tau_n \rangle$, we have written $H[a \mapsto desc]$ to denote a heap updated with $(a, fnm_i) \mapsto default(\tau_i)$, where $default(\mathsf{int}) = 0$ and $default(\mathsf{ref}) = null$. For the free instruction, $H \backslash \langle a, desc \rangle$ denotes the removal of all elements with keys (a, fnm_i) in H.

$$\frac{code[pc] = \mathsf{iconst}\ z}{\langle code, S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, z :: S, L, pc + 1 \rangle}$$

$$\frac{code[pc] = \mathsf{ibinop}\ op}{\langle code, z_1 :: z_2 :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, (\llbracket op \rrbracket\ z_1\ z_2) :: S, L, pc + 1 \rangle}$$

$$\frac{code[pc] = \mathsf{pop}}{\langle code, z :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L, pc + 1 \rangle} \qquad \frac{code[pc] = \mathsf{load}\ n \qquad L[n] = v}{\langle code, S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, v :: S, L, pc + 1 \rangle}$$

$$\frac{code[pc] = \mathsf{store}\ n}{\langle code, v :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L[n \mapsto v], pc + 1 \rangle}$$

$$\frac{code[pc] = \mathsf{aconst\_null}}{\langle code, S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, null :: S, L, pc + 1 \rangle}$$

$$\frac{code[pc] = \mathsf{binarycmp}\ cmp\ offset \qquad z_1\ \llbracket cmp \rrbracket\ z_2}{\langle code, z_1 :: z_2 :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L, offset \rangle}$$

$$\frac{code[pc] = \mathsf{binarycmp}\ cmp\ offset \qquad \neg(z_1\ \llbracket cmp \rrbracket\ z_2)}{\langle code, z_1 :: z_2 :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L, pc + 1 \rangle}$$

$$\frac{code[pc] = \mathsf{unarycmp}\ cmp\ offset \qquad z\ \llbracket cmp \rrbracket\ 0}{\langle code, z :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L, offset \rangle}$$

$$\frac{code[pc] = \mathsf{unarycmp}\ cmp\ offset \qquad \neg(z\ \llbracket cmp \rrbracket\ 0)}{\langle code, z :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L, pc + 1 \rangle}$$

$$\frac{code[pc] = \mathsf{ifnull}\ offset \qquad a = null}{\langle code, a :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L, offset \rangle} \qquad \frac{code[pc] = \mathsf{ifnull}\ offset \qquad a \neq null}{\langle code, a :: S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L, pc + 1 \rangle}$$

$$\frac{code[pc] = \mathsf{goto}\ offset}{\langle code, S, L, pc \rangle \xrightarrow{\mathsf{frm}} \langle code, S, L, offset \rangle}$$

Figure 2: Intra-frame Operational Semantics Rules

$$\frac{code[pc] = \mathsf{new}\ desc \qquad (\forall fnm.\ (a, fnm) \notin H)}{\langle code, S, L, pc\rangle, H \xrightarrow{\mathsf{mut}} \langle code, a :: S, L, pc + 1\rangle, H[a \mapsto desc], e}$$

$$\frac{code[pc] = \mathsf{getfield}\ fnm \qquad H[a, fnm] = v}{\langle code, a :: S, L, pc\rangle, H \xrightarrow{\mathsf{mut}} \langle code, v :: S, L, pc + 1\rangle, H, e}$$

$$\frac{code[pc] = \mathsf{putfield}\ fnm}{\langle code, a :: v :: S, L, pc\rangle, H \xrightarrow{\mathsf{mut}} \langle code, S, L, pc + 1\rangle, H[(a, fnm) \mapsto v], e}$$

$$\frac{code[pc] = \mathsf{free}\ desc}{\langle code, a :: S, L, pc\rangle, H \xrightarrow{\mathsf{mut}} \langle code, S, L, pc + 1\rangle, H \backslash \langle a, desc\rangle, e}$$

$$\frac{code[pc] = \mathsf{consume}\ r}{\langle code, S, L, pc\rangle, H \xrightarrow{\mathsf{mut}} \langle code, S, L, pc + 1\rangle, H, r}$$

Figure 3: Heap and Resource Mutating Operational Semantics Rules

$$\frac{f \xrightarrow{\mathsf{frm}} f'}{\langle r, H, f :: fs\rangle \xrightarrow{prg} \langle r, H, f' :: fs\rangle} \qquad \frac{f, H \xrightarrow{\mathsf{mut}} f', H', r_c}{\langle r, H, f :: fs\rangle \xrightarrow{prg} \langle r \cdot r_c, H', f' :: fs\rangle}$$

$$\frac{code[pc] = \mathsf{return}}{\langle r, H, \langle code, v :: S, L, pc\rangle :: \langle code', S', L', pc'\rangle :: fs\rangle \xrightarrow{prg} \langle r, H, \langle code', v :: S', L', pc'\rangle :: fs\rangle}$$

$$\frac{code[pc] = \mathsf{call}\ pname \qquad prg[pname] = code'}{\langle r, H, \langle code, args +\!\!+ S, L, pc\rangle :: fs\rangle \xrightarrow{prg} \langle r, H, \langle code', [], \ulcorner args \urcorner, 0\rangle :: \langle code, S, L, pc + 1\rangle :: fs\rangle}$$

Figure 4: Small-step Operational Semantics Rules

Note that all the rules in the $\xrightarrow{\mathsf{mut}}$ relation apart from the consume instruction consume no resources: e is the identity element of our resource monoid.

A state of the full virtual machine is a tuple $\langle r, H, fs\rangle \in \mathsf{State}$, where r is the resource consumed to this point, h is the current heap, and fs is a list of activation frames. The small-step operational semantics of the full machine for some program prg is given by a relation $\xrightarrow{prg} \subseteq \mathsf{State} \times \mathsf{State}$ which incorporates the $\xrightarrow{\mathsf{frm}}$ and $\xrightarrow{\mathsf{mut}}$ relations and also describes how the call and return instructions manipulate the stack of activation frames. The rules defining this relation are presented in Figure 4. In the rule for the instruction call, the operation $+\!\!+$ denotes list concatenation, $[]$ denotes the empty list and $\ulcorner - \urcorner$ denotes the translation of a list to a finite map from natural numbers in the obvious way.

Finally, we use the predicate $s \downarrow H, r, v$ to indicate when a return instruction is to be executed and there is only one activation frame on the stack. In this case execution of the

program terminates. The H, r and v are the final heap, the total consumed resources and the return value of the program respectively.

3.3. **Assertions.** Every procedure *pname* in the program is annotated with a precondition and a post-condition. To allow for variables that are universally quantified over both the pre- and post-condition we make use of Coq's dependent types to augment procedure specifications with a specific "environment" type. A procedure specification is a dependent triple:

$$\langle \mathbb{E} : \mathsf{Type}, P \subseteq \mathbb{E} \times \mathbb{V}^* \times \mathbb{H} \times \mathcal{R}, Q \subseteq \mathbb{E} \times \mathbb{V}^* \times \mathbb{H} \times \mathcal{R} \times \mathbb{V} \rangle$$

Preconditions P are predicates over $\mathbb{E} \times \mathbb{V}^* \times \mathbb{H} \times \mathcal{R}$: environments, lists of arguments to the procedure and the heap and available resource at the start of the procedure's execution. Post-conditions are predicates over $\mathbb{E} \times \mathbb{V}^* \times \mathbb{H} \times \mathcal{R} \times \mathbb{V}$: environments, argument lists and the heap, remaining consumable resource and return value.

Intermediate assertions in our program logic are predicates over $\mathbb{E} \times \mathbb{V}^* \times \mathbb{H} \times \mathcal{R} \times \mathbb{V}^* \times (\mathbb{N} \rightharpoonup \mathbb{V})$: environments, argument lists, the heap, remaining consumable resource and the current operand stack and local variable store. Intermediate assertions are the assertions that are attached to every instruction in the body of a procedure by our program logic and specify a sufficient precondition for safely executing that instruction and all of its successors.

Note that each of the three different types of assertions talks about the *remaining* consumable resources available to the program, not the resources that have already been consumed.

3.4. **Program Logic.** A proof that a given procedure's implementation *code* matches its specification $\langle \mathbb{E}, P, Q \rangle$ consists of a map C from instruction offsets in *code* to assertions such that:

(1) Every instruction's assertion is suitable for that instruction: for every instruction offset i in *code*, there exists an assertion A such that $C \vdash_Q \{A\} \Rightarrow i{:}code[i]$, and $C[i]$ implies A. Figure 5 gives the definition of the judgement $C \vdash_Q \{A\} \Rightarrow i{:}\iota$ for a selected subset of the instructions ι. The post-condition Q is used for the case of the return instruction. We explain the definition of this judgement in more detail below.
(2) The precondition implies the assertion for the first instruction: for all environments $env \in \mathbb{E}$, arguments *args*, heaps H and consumable resources r, we have

$$P(e, args, H, r) \Rightarrow C[0](env, args, H, r, [], \ulcorner args \urcorner)$$

where $[]$ denotes the empty operand stack, and $\ulcorner - \urcorner$ maps lists of values to finite maps from naturals to values, as in the operational semantics in Figure 4.

When condition 1 holds, we write this as $C \vdash code : Q$, indicating that the procedure implementation *code* has a valid proof C for the post-condition Q.

The rules in Figure 5 are an illustrative subset of the rules of the program logic. The rule for iconst merely states that the precondition for executing this instruction is the precondition of the next instruction with the specified integer pushed on to the stack. The rules for most of the other intra-frame, non-mutating instructions are similar. Slightly different are the rules for the conditional instructions, for example ifnull; these make use of logical conjunction to make sure that both possible outcomes of the conditional have their preconditions satisfied.

$$C \vdash_Q \{\lambda(env, args, r, H, S, L).\ C[i+1](env, args, r, H, z :: S, L)\} \Rightarrow i{:}\mathsf{iconst}\ z$$

$$C \vdash_Q \left\{ \begin{array}{l} \lambda(env, args, r, H, S, L). \\ \quad \forall a, S'. S = a :: S' \Rightarrow \\ \qquad (a \neq null \Rightarrow C[i+1](env, args, r, H, S', L)) \wedge \\ \qquad (a = null \Rightarrow C[n](env, args, r, H, S', L)) \end{array} \right\} \Rightarrow i{:}\mathsf{ifnull}\ n$$

$$C \vdash_Q \left\{ \begin{array}{l} \lambda(env, args, r, H, S, L). \\ \quad \forall a, v, S'. \\ \qquad S = a :: v :: S' \wedge \\ \qquad (a, fnm) \in H \wedge \\ \qquad C[i+1](env, args, r, H[(a, fnm) \mapsto v], S', L) \end{array} \right\} \Rightarrow i{:}\mathsf{putfield}\ fnm$$

$$C \vdash_Q \left\{ \begin{array}{l} \lambda(env, args, r, H, S, L). \\ \quad \exists r'.\ r_c \cdot r' \sqsubseteq r \wedge C[i+1](env, args, r', H, S, L) \end{array} \right\} \Rightarrow i{:}\mathsf{consume}\ r_c$$

$$C \vdash_Q \left\{ \begin{array}{l} \lambda(env, args, r, H, S, L). \\ \quad \forall args'\ S'. S = args' +\!\!+ S' \Rightarrow \\ \qquad \exists env' \in \mathbb{E}_p, (H_1, r_1), (H_2, r_2). \\ \qquad\quad R(H_1, r_1)(H_2, r_2)(H, r) \wedge \\ \qquad\quad P_p(env', args', H_1, r_1) \wedge \\ \qquad\quad \forall v, (H_1', r_1'). \\ \qquad\qquad H_1' \# H_2 \Rightarrow \\ \qquad\qquad Q_p(env', args', H_1', r_1', v) \Rightarrow \\ \qquad\qquad C[i+1](env, args', r_1' \cdot r_2, H_1' \uplus H_2, v :: S', L) \end{array} \right\} \Rightarrow i{:}\mathsf{call}\ p$$

$$C \vdash_Q \{\lambda(env, args, r, H, S, L). \forall v, S'.\ S = v :: S' \Rightarrow Q(env, args, r, H, v)\} \Rightarrow i{:}\mathsf{return}$$

Figure 5: Program Logic Rules (Extract)

The rules for putfield and consume demonstrate heap and consumable resource consumption respectively. For putfield, the heap is judged to be satisfactory if it contains the address and field that are to be mutated. For resource consumption, we must "subtract" the consumed resource from the current resource that has been supplied. If the desired resource is not present then this instruction's precondition will not hold.

The rule for the call instruction demonstrates how the heap and the consumable resources are dealt with similarly for the purposes of a frame rule. The current heap and consumable resources are split into two parts (H_1, r_1) and (H_2, r_2), with the first part being handed off to the callee for it to be mutated. Finally, the precondition of the return instruction is directly derived from the post-condition of the current procedure.

3.5. **Soundness.** Soundness for the program logic is stated as the preservation of a safety invariant by every step of the virtual machine. We define safety for activation frames, frame stacks and whole machine states, each building on the last.

We say that an activation frame is safe if there is a proof for the code being executed in the frame such that the requirements of the next instruction to be executed are satisfied. Formally, a frame $f = \langle code, S, L, pc \rangle$ is safe for environment $env \in \mathbb{E}$, arguments $args$, heap H, resource r and post-condition Q, written $safeFrame(env, f, H, r, args, Q)$ if[3]:

(1) There exists a certificate C such that $C \vdash code : Q$;
(2) $C[pc]$ exists and $C[pc](env, args, r, H, S, L)$ holds.

Safety of activation frames is preserved by steps in the virtual machine:

Lemma 3.1 (Intra-frame safety preservation)**.**

(1) *If*
   (a) $safeFrame(env, f, H, r, args, Q)$
   (b) $f \xrightarrow{\mathsf{frm}} f'$
   then $safeFrame(env, f', H, r, args, Q)$.
(2) *If*
   (a) $safeFrame(env, f, H_1, r, args, Q)$
   (b) $H_1 \# H_2$ *and* $H_1 \uplus H_2 = H$
   (c) $f, H \xrightarrow{\mathsf{mut}} f', H', r_c$
   then there exists H_1' *and* r' *such that*
   (a) $H_1' \# H_2$ *and* $H_1' \uplus H_2 = H'$
   (b) $r_c \cdot r' \sqsubseteq r$
   (c) $safeFrame(env, f', H_1', r', args, Q)$.

The second part of this lemma states that if we take a step that mutates the heap or consumes some resource, and the activation frame has been certified as safe for a sub-part of the heap, then the rest of the heap—H_2—is unaffected by a single step of execution in this activation frame, and the new state of the activation frame is safe for the mutated heap and new amount of consumable resources.

Remark 3.2. We pause for a moment to consider the relationship between our program logic and traditional Separation Logic. The second part of the previous lemma effectively states that execution steps for mutating instructions are *local*: for any other piece of heap that is present but not mentioned in its precondition, the execution of a mutating instruction will not affect it. This is usually expressed in Separation Logic by the frame rule that states if we know that $\{P\}C\{Q\}$ holds, then $\{P * R\}C\{Q * R\}$ holds for any other resource assertion R. We do not have an explicit frame rule in our program logic; application of the rule is implicit in the rule for the call instruction (so, conflatingly, the frame rule is applied for new activation frames). We do not have access to the frame rule in order to modularly reason about the internals of each procedure, e.g. local reasoning about individual loops. This is partially a consequence of the unstructured nature of the bytecode that we are working with. It has not been a hindrance in small examples that we have verified so far, but may well become so in larger procedures with multiple loops that need invariants—see Section 2.2 for an example. In such cases it may be useful to layer a hierarchical structure, matching the loops or other sub-program structure, on top of the unstructured bytecode in order to apply frame rules and facilitate local reasoning inside procedures.

[3]In this definition, and all the later ones in this section, we have omitted necessary assertions about well-typedness of the stack, local variables and the heap because they would only clutter our presentation.

We have now handled all the instructions except the call and return instructions that create and destroy activation frames. To state soundness of our program logic for these we need to define what it means for a stack of activation frames to be safe. Intuitively, a stack of activation frames is a bridge between the overall arguments $\mathit{args}_{\mathit{top}}$ and post-condition Q_{top} for the program and the arguments $\mathit{args}_{\mathit{cur}}$ and post-condition Q_{cur} for the current activation frame, with respect to the current heap H and available consumable resources r, such that, when the current activation frame finishes, its calling frame on the top of the stack is safe. We write this as $\mathit{safeStack}_{\mathbb{E}_{\mathit{cur}},\mathbb{E}_{\mathit{top}}}(\mathit{fs}, H, r, \mathit{env}_{\mathit{cur}}, \mathit{args}_{\mathit{cur}}, Q_{\mathit{cur}}, \mathit{env}_{\mathit{top}}, \mathit{args}_{\mathit{top}}, Q_{\mathit{top}})$. The types $\mathbb{E}_{\mathit{cur}}$ and $\mathbb{E}_{\mathit{top}}$ refer to the environment types for the current and top-level procedures respectively, and $\mathit{env}_{\mathit{cur}} \in \mathbb{E}_{\mathit{cur}}$, $\mathit{env}_{\mathit{top}} \in \mathbb{E}_{\mathit{top}}$ are the specific elements being used.

Accordingly, we say that the empty frame stack is safe when $r = e$, $H = \mathsf{emp}$, $\mathit{env}_{\mathit{cur}} = \mathit{env}_{\mathit{top}}$, $\mathit{args}_{\mathit{cur}} = \mathit{args}_{\mathit{top}}$ and $Q_{\mathit{cur}} = Q_{\mathit{top}}$. A non-empty frame stack $\mathit{fs} = \langle \mathit{code}, S, L, \mathit{pc}\rangle :: \mathit{fs}'$ is safe when there exist (H_1, r_1), (H_2, r_2), env, args, Q and C, A such that:

(1) $R(H_1, r_1)(H_2, r_2)(H, r)$;
(2) The code is certified: $C \vdash \mathit{code} : Q$;
(3) The next instruction to be executed has precondition A: $C[\mathit{pc}] = A$;
(4) When the callee returns, the instruction's precondition will be satisfied: for all $v \in \mathbb{V}, H_2', r_2'$ such that $H_2' \# H_1$ and $Q_{\mathit{cur}}(\mathit{env}_{\mathit{cur}}, \mathit{args}_{\mathit{cur}}, H_2', r_2', v)$ holds, $A(\mathit{env}, \mathit{args}, r_2' \cdot r_1, H_2' \uplus H_1, v :: S, L)$ holds as well.
(5) The rest of the frame stack fs will be safe when this activation frame returns: $\mathit{safeStack}(\mathit{fs}, H_2, r_2, \mathit{env}, \mathit{args}, Q, \mathit{env}_{\mathit{top}}, \mathit{args}_{\mathit{top}}, Q_{\mathit{top}})$.

Note how the *safeStack* predicate divides up the heap and consumable resource between the activation frames on the call stack; each frame hands a piece of its heap and consumable resource off to its callees to use. This mirrors the formulation of the rule for call in the program logic in Figure 5.

Finally, we say that a state $s = \langle r_c, H, \mathit{fs}\rangle$ is safe for environment env, arguments args, post-condition Q and maximum resource r_{max}, written $\mathit{safeState}(s, \mathit{env}, \mathit{args}, Q, r_{\mathit{max}})$, if:

(1) there exists an r_{future} such that $r_c \cdot r_{\mathit{future}} \sqsubseteq r_{\mathit{max}}$; and also
(2) r_{future} and H split into (H_1, r_1) and (H_2, r_2), i.e. $R(H_1, r_1)(H_2, r_2)(H, r_{\mathit{future}})$;
(3) there exists at least one activation frame: $\mathit{fs} = f :: \mathit{fs}'$ and environment $\mathit{env}_{\mathit{cur}}$, arguments $\mathit{args}_{\mathit{cur}}$ and post-condition Q_{cur}; such that
(4) $\mathit{safeFrame}(f, H_1, r_1, \mathit{env}_{\mathit{cur}}, \mathit{args}_{\mathit{cur}}, Q_{\mathit{cur}})$; and
(5) $\mathit{safeStack}(\mathit{fs}, H_2, r_2, \mathit{env}_{\mathit{cur}}, \mathit{args}_{\mathit{cur}}, Q_{\mathit{cur}}, \mathit{env}, \mathit{args}, Q)$.

The key point in the definition of *safeState* is that the assertions of the program logic talk about the resources that will be consumed in the *future* of the program's execution. Safety for a state says that when we combine the future resource requirements r_{future} with resources that have been consumed in the past, r_c, then the total is less than the total resources r_{max} that are allowed for the execution.

Theorem 3.3 (Soundness)**.** *Assume that all the procedures in prg match their specifications.*

(1) *If*
   (a) $\mathit{safeState}(s, \mathit{env}, \mathit{args}, Q, r_{\mathit{max}})$*; and*
   (b) $s \xrightarrow{\mathit{prg}} s'$
   then $\mathit{safeState}(s', \mathit{env}, \mathit{args}, Q, r_{\mathit{max}})$.
(2) *If* $\mathit{safeState}(s, \mathit{env}, \mathit{args}, Q, r_{\mathit{max}})$ *and* $s \downarrow H, r, v$*, then there exists an* r' *such that* $Q(\mathit{env}, \mathit{args}, H, r', v)$ *and* $r \cdot r' \sqsubseteq r_{\mathit{max}}$.

In the halting case in this theorem, the existentially quantified resource r' indicates the resources that the program still had available at the end of its execution. We are also guaranteed that when the program halts, the total resource that it has consumed will be less than the fixed maximum r_{max} that we have set, and moreover, by item 1 of the theorem, this bound has been observed at every step of the computation.

Remark 3.4. Though we assumed it above, the proof of soundness of the program logic does not require that the monoid of resources is commutative. This opens the way to considering non-commutative notions of resource, such as traces. However, constructing a usable proof-theory for a mixed commutative/non-commutative notion of resource (i.e. heaps and a putative non-commutative consumable resource) is hard. Also, the resource acquisition variant of the program logic that we describe in the next section requires commutativity.

3.6. **Allowing for Resource Acquisition.** The operational semantics and program logic described above assume a fixed total amount of resource that may be consumed by the program. The precondition of the main procedure of the program specifies the amount of resource that will be required for the entire run. In this section we consider the changes necessary to support dynamic but fallible acquisition of resources via a special acquire instruction. We provide an example use of this additional capability in Section 3.7.

We assume that there is a capricious environment that entertains requests for additional resources from programs. Requests may be granted or denied. A program may request an additional resource using a special acquire instruction. To make things interesting, and to allow for the example in the next section, we make the resource requested by acquire dynamic and correspondingly modify consume, removing its static argument:

$$\iota ::= \ldots \mid \mathsf{consume} \mid \mathsf{acquire}$$

For both instructions, we assume that the resource being consumed or requested is indicated by an integer value on the stack. For the acquire instruction the intended semantics is that if the request is granted then a 1 is pushed on to the operand stack, otherwise a 0 is pushed.

For the operational semantics, we modify the relation $\xrightarrow{\mathsf{mut}}$ to have an additional "acquired resources" component:

$$f, H \xrightarrow{\mathsf{mut}} f, H, r_{consumed}, r_{acquired}$$

For each of the existing heap mutating instructions except consume, the acquired resource is equal to the unit of the resource monoid, e. The operational semantics of consume is replaced by the following rule to reflect the dynamic resource identification:

$$\frac{code[pc] = \mathsf{consume}}{\langle code, z :: S, L, pc\rangle, H \xrightarrow{\mathsf{mut}} \langle code, S, L, pc+1\rangle, H, \mathrm{res}(z), e}$$

where we assume the existence of a function $\mathrm{res} : \mathbb{Z} \to \mathcal{R}$ that names certain consumable resources in $\mathcal{R}$ by integers. We do not assume that res is a monoid homomorphism. Two new operational semantics rules are added for the acquire instruction, for the two possible

outcomes of requesting more resource:

$$\frac{code[pc] = \mathsf{acquire}}{\langle code, z :: S, L, pc\rangle, H \xrightarrow{\mathsf{mut}} \langle code, 1 :: S, L, pc+1\rangle, H, e, \mathrm{res}(z)}$$

$$\frac{code[pc] = \mathsf{acquire}}{\langle code, z :: S, L, pc\rangle, H \xrightarrow{\mathsf{mut}} \langle code, 0 :: S, L, pc+1\rangle, H, e, e}$$

States of the virtual machine are modified to be four-tuples $\langle r_{con}, r_{tot}, H, fs\rangle$ of consumed resources, total allowed resources, current heap and current activation frame stack. The invariant that is now to be maintained is that the r_{con} is always less than or equal to r_{tot}. The only rule from Figure 4 that is modified (apart from threading through the unchanged r_{tot}) is the rule incorporating $\xrightarrow{\mathsf{mut}}$ into the relation:

$$\frac{f, H \xrightarrow{\mathsf{mut}} f', H', r_c, r_a}{\langle r, r_{tot}, H, f :: fs\rangle \xrightarrow{prg} \langle r \cdot r_c, r_{tot} \cdot r_a, H', f' :: fs\rangle}$$

The halting predicate is extended to a five-place relation $s \downarrow H, r_{con}, r_{tot}, v$, where r_{con} is the total consumed resource of the execution and r_{tot} is the total acquired resource.

Perhaps surprisingly, the assertions (pre- and post-conditions and instruction level assertions) are left unchanged, as are the rules of the logic from Figure 5 for the original set of instructions. We modify the rule for resource consumption to take into account the new dynamic nature:

$$C \vdash_Q \left\{ \begin{array}{l} \lambda(env, args, r, H, S, L). \\ \quad \forall z, S'.\ S = z :: S' \Rightarrow \exists r'. \\ \qquad \mathrm{res}(z) \cdot r' \sqsubseteq r \wedge C[i+1](env, args, r', H, S', L) \end{array} \right\} \Rightarrow i{:}\mathsf{consume}$$

and we add a new rule for resource acquisition:

$$C \vdash_Q \left\{ \lambda(env, args, r, H, S, L). \quad \begin{array}{l} \forall z, S'.\ S = z :: S' \Rightarrow \\ \quad C[i+1](env, args, r \cdot \mathrm{res}(z), H, 1 :: S', L) \\ \quad \wedge\ C[i+1](env, args, r, H, 0 :: S', L) \end{array} \right\} \Rightarrow i{:}\mathsf{acquire}$$

This exactly mirrors the pair of operational semantics rules. An acquire instruction is safe to run if it is safe the execute the next instruction either with additional available consumable resource and a 1 on the stack, or with no additional consumable resource and a 0 on the stack.

The definition of safe frame remains unchanged from above, and the statement of the second item of Lemma 3.1 is adjusted to account for the possibility of additional resources being acquired:

Lemma 3.5 (Intra-frame safety preservation (Resource Acquisition Variant))**.**

(2) *If*
  (a) $safeFrame(env, f, H_1, r, args, Q)$
  (b) $H_1 \# H_2$ *and* $H_1 \uplus H_2 = H$
  (c) $f, H \xrightarrow{\mathsf{mut}} f', H', r_c, r_a$

  then there exists H_1' *and* r' *such that*
  (a) $H_1' \# H_2$ *and* $H_1' \uplus H_2 = H'$
  (b) $r_c \cdot r' \sqsubseteq r \cdot r_a$
  (c) $safeFrame(env, f', H_1', r', args, Q)$.

The definition of the *safeStack* predicate remains the same as before, and the *safeState* predicate is modified to be of the form $safeState(s, env, args, Q)$: the original r_{max} argument is taken to be the r_{tot} from the state. The key safety property, as before, is that the consumed resources, plus the future resources, is less than or equal to the total allowed resources. But now the total resources allowed may be increased during execution.

The new version of Theorem 3.3 is as follows:

Theorem 3.6 (Soundness (Resource Acquisition Variant))**.** *Assume that all the procedures in prg match their specifications.*

(1) *If*
   (a) $safeState(s, env, args, Q)$*; and*
   (b) $s \xrightarrow{prg} s'$
   then $safeState(s', env, args, Q)$*.*
(2) *If* $safeState(s, env, args, Q)$ *and* $s \downarrow H, r_{con}, r_{tot}, v$*, then there exists an* r' *such that* $Q(env, args, H, r', v)$ *and* $r_{con} \cdot r' \sqsubseteq r_{tot}$*.*

This theorem gives a comparable guarantee to Theorem 3.3, in that the (now dynamic) resource bound is respected at every step of the computation and at the end of execution. By inspection of the operational semantics, we can see that the resource bound may only be increased by successful execution of an acquire instruction.

3.7. **Resource Acquisition Example.** To illustrate the utility of the language and logic extended with resource acquisition, we take the example of *block booking* as presented by Aspinall, Maier and Stark [4]. The scenario is that an application running on a mobile device has a list of telephone numbers that it wants to send SMS messages to. Permission must be sought from the user to send these messages, since sending incurs a monetary cost. It is not necessarily convenient to request permission from the user when the message is to be sent, so permission is requested in advance. We use the program logic extended with resource acquisition to bridge the gap between the acquisition of permission and its consumption, ensuring that no attempt is made to perform an operation that has not been authorised. In this scenario, the acquire instruction is implemented by actually requesting permission from the user.

Figure 6 shows the code for a method, `requestPermissions`, that loops through a list of telephone numbers, requesting permission for each one. We assume that the calls to the method `acquire` are compiled to instances of the acquire instruction. We have taken the liberty of assuming a value type of telephone numbers rather than re-using the `int` type. The result of the resource acquisition attempt is stored in the `permission` field of the record for that telephone number.

The specification of the precondition of `requestPermissions` is simply that there is a proper linked list on the heap, using the following inductively defined predicate:

$$\mathsf{lseg}(x, y) \equiv (x = y \wedge \mathsf{emp}) \vee (\exists n, p, z.\ [x \overset{\mathsf{number}}{\mapsto} n] * [x \overset{\mathsf{permission}}{\mapsto} p] * [x \overset{\mathsf{next}}{\mapsto} z] * \mathsf{lseg}(z, y))$$

In the post-condition, we make use of another inductively defined predicate that states the resources available, dependent on the value of the `permission` field:

$$\begin{aligned}\mathsf{lseg}^{\mathsf{send}}(x, y) \equiv\ & (x = y \wedge \mathsf{emp}) \\ & \vee\ (\exists n, z.\ [x \overset{\mathsf{number}}{\mapsto} n] * [x \overset{\mathsf{permission}}{\mapsto} 1] * R^{\mathsf{send}(n)} * [x \overset{\mathsf{next}}{\mapsto} z] * \mathsf{lseg}(z, y)) \\ & \vee\ (\exists n, z.\ [x \overset{\mathsf{number}}{\mapsto} n] * [x \overset{\mathsf{permission}}{\mapsto} 0] * [x \overset{\mathsf{next}}{\mapsto} z] * \mathsf{lseg}(z, y))\end{aligned}$$

```
class PhoneNumberNode {
     public PhoneNumber     number;
     public boolean         permission;
     public PhoneNumberNode next;
}

...

public static void requestPermissions (PhoneNumberNode phoneNumber) {
     while (phoneNumber != null) {
          phoneNumber.permission = acquire (phoneNumber.number);
          phoneNumber = phoneNumber.next;
     }
}
```

Figure 6: Resource Acquisition Example

Thus when the `permission` field has the value 1, the list node is associated with the permission to send to that telephone number; and when the `permission` field has the value 0, no such associated permission is available. We revisit this kind of conditionally-resource-carrying predicate in Section 4.2.

Re-interpreting the consume instruction as sending an SMS message, a loop that sends messages to all telephone numbers that the user has approved now looks much like the simple list iteration example from the introduction, augmented with a dynamic check to ensure that permission has been sought and received.

4. Deep Assertion Logic

In the previous section we described a program logic but remained agnostic as to the exact form of the assertions save that they must be predicates over certain domains. This shallow approach makes the statement and soundness proof easier, but inhibits discussion of actual specifications and proofs in the logic. In this section we show how a combination of two variants of the logic of Bunched Implications (BI) [20, 22] can be used to provide a syntax for assertions in our program logic. We combine boolean BI with affine intuitionistic BI, for describing heaps and consumable resources respectively.

4.1. **Syntax and Semantics.** We made use of three different types of assertion for the program logic: procedure pre- and post-conditions, and intermediate assertions within procedures. These all operate on heaps and consumable resources and the arguments to the current procedure, but differ in whether they talk about return values or the operand stack and local variables. To deal with these differences we assume that we have a set of terms in our logic, ranged over by $t, t_1, t_2, ...$, that at least includes logical variables and a constant *null* for representing the null reference, and also variables for representing the current procedure arguments, the return value and the operand stack and local variables as appropriate.

$$
\begin{array}{ll}
\eta, x \models \top & \text{iff always} \\
\eta, x \models t_1 \bowtie t_2 & \text{iff } [\![t_1]\!]_\eta \bowtie [\![t_2]\!]_\eta \\
\eta, x \models \mathsf{emp} & \text{iff } x = (H, r) \text{ and } H = \{\} \\
\eta, x \models [t_1 \overset{\mathsf{f}}{\mapsto} t_2] & \text{iff } x = (H, r) \text{ and } H = \{([\![t_1]\!]_\eta, \mathsf{f}) \mapsto [\![t_2]\!]_\eta\} \\
\eta, x \models R_{r_i} & \text{iff } x = (H, r) \text{ and } r_i \sqsubseteq r \text{ and } H = \{\} \\
\eta, x \models \phi_1 \wedge \phi_2 & \text{iff } \eta, x \models \phi_1 \text{ and } \eta, x \models \phi_2 \\
\eta, x \models \phi_1 \vee \phi_2 & \text{iff } \eta, x \models \phi_1 \text{ or } \eta, x \models \phi_2 \\
\eta, x \models \phi_1 * \phi_2 & \text{iff exists } y, z. \text{ st. } Ryzx \text{ and } \eta, y \models \phi_1 \text{ and } \eta, z \models \phi_2 \\
\eta, x \models \phi_1 \to \phi_2 & \text{iff for all } y. \text{ if } x \sqsubseteq y \text{ and } \eta, y \models \phi_1 \text{ then } \eta, y \models \phi_2 \\
\eta, x \models \phi_1 \mathrel{-\!\!*} \phi_2 & \text{iff for all } y, z. \text{ if } Rxyz \text{ and } \eta, y \models \phi_1 \text{ then } \eta, z \models \phi_2 \\
\eta, x \models \forall v.\phi & \text{iff for all } a, \eta[v \mapsto a], x \models \phi \\
\eta, x \models \exists v.\phi & \text{iff exists } a, \eta[v \mapsto a], x \models \phi
\end{array}
$$

Figure 7: Semantics of assertions

Formulae are built from at least the following constructors:

$$
\begin{array}{rcl}
\phi & ::= & \top \mid t_1 \bowtie t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \to \phi_2 \mid \mathsf{emp} \mid \phi_1 * \phi_2 \mid \phi_1 \mathrel{-\!\!*} \phi_2 \mid \forall x.\phi \mid \exists x.\phi \\
& & \mid [t_1 \overset{\mathsf{f}}{\mapsto} t_2] \mid R_r \mid \ldots
\end{array}
$$

Where $\bowtie \in \{=, \neq\}$. We can also add inductively defined predicates as needed, see Section 4.2 below. The only non-standard formula with respect to Separation Logic is R_r which represents the presence of some consumable resource r. The semantics of the assertion logic is given in Figure 7 as a relation $\models$ between environments and heap/consumable resource pairs and formulae. We assume a sensible semantics $[\![\cdot]\!]_\eta$ for terms in a given environment.

As a consequence of having an ordering on consumable resources, and our chosen semantics of emp, $*$ and $\mathrel{-\!\!*}$, our logic contains affine intuitionistic BI as a sub-logic for reasoning purely about consumable resources.

Proposition 4.1. *If ϕ is a propositional BI formula with only R_r as atoms, then $r \models_{\mathrm{bi}} \phi$ iff $\eta, (r, h) \models \phi$.*

4.2. **Inductively Defined Shape and Resource Predicates.** We now present some inductively defined predicates that demonstrate how heap-resident data structures may have resources associated with their nodes. We have introduced the resource-aware lseg predicate that describes a segment of a list with a resource associated to every element:

$$\mathsf{lseg}(r, x, y) \equiv (x = y \wedge \mathsf{emp}) \vee (\exists d, z.\ [x \overset{\mathsf{data}}{\mapsto} d] * [x \overset{\mathsf{next}}{\mapsto} z] * R^r * \mathsf{lseg}(r, z, y))$$

An alternative that we made use of in the block booking example in Section 3.7 is to only demand resources when the element data satisfies some predicate, for example when the integer stored in the node is not equal to zero:

$$\mathsf{lseg}^{\neq 0}(r, x, y) \equiv (x = y \wedge \mathsf{emp}) \vee (\exists d, z.\ [x \overset{\mathsf{data}}{\mapsto} d] * [x \overset{\mathsf{next}}{\mapsto} z] * (d \neq 0 \to R^r) * \mathsf{lseg}^{\neq 0}(r, z, y))$$

This kind of specification allows the conditional resource property to be specified locally within the list structure. If we were attempting explicit resource accounting using sized

predicates, we would be forced to reflect the whole list into the logic and state the resource requirement in terms of the number of non-zero elements:

$$\mathsf{lseg}(l, x, y) \wedge r = \mathsf{length}(\mathsf{filter}(\lambda x.x \neq 0, l))$$

Reasoning with such global list properties is obviously much harder than locally reasoning about each individual node as it is processed by the program.

The amortised approach is not limited to reasoning purely about singly-linked lists, the standard doubly-linked list and tree predicates of Separation Logic can be easily augmented with local resource annotations (we have omitted the data components of these predicates to save space):

$$\mathsf{dlseg}(r, p, x, y) \equiv (x = y \wedge \mathsf{emp}) \vee (\exists z.\ [x \overset{\mathsf{next}}{\mapsto} z] * [x \overset{\mathsf{prev}}{\mapsto} p] * R^r * \mathsf{dlseg}(r, x, z, y))$$

$$\begin{aligned}\mathsf{tree}(r, x) \equiv\ & (x = \mathit{null} \wedge \mathsf{emp}) \\ & \vee (\exists y, z.\ [x \overset{\mathsf{left}}{\mapsto} y] * [x \overset{\mathsf{right}}{\mapsto} z] * R^r * \mathsf{tree}(y) * \mathsf{tree}(z))\end{aligned}$$

These predicates all describe resources that are linear in proportion to the sizes of the data structures. Using the clever technique of Hoffmann and Hofmann [14], we can also present lists with associated resources that are polynomially proportional to the length of the list, by exploiting the presentation of polynomials using binomial coefficients. In their system, lists are annotated with resources that are lists of rational numbers $\langle p_1, ..., p_n \rangle$. The idea is that a list of length n annotated with such a list has $\sum_{i=1}^{k} \binom{n}{i} p_i$ associated resource. We can give such lists as an inductively defined predicate in our logic:

$$\mathsf{lseg}(\overrightarrow{p}, x, z) \equiv (x = z \wedge \mathsf{emp}) \vee (\exists y.\ [x \overset{\mathsf{next}}{\mapsto} y] * R^{p_1} * \mathsf{lseg}(\lhd(\overrightarrow{p}), y, z))$$

where $\lhd(\overrightarrow{p}) = (p_1 + p_2, p_2 + p_3, ..., p_{k-1}, p_k)$ is the additive shift of a resource annotation as defined by Hoffmann and Hofmann.

4.3. **Heap and Resource Separation.** In the logic of Section 4.1, we only made use of one kind of separating conjunction, $\phi_1 * \phi_2$, that separates both heaps and consumable resources. This allows the tight integration of the heap shapes of various data structures and consumable resources as shown in the previous section. Evidently, there are two other possible combinations that allow sharing of heap or resources. For example, separation of resources, but sharing of heap:

$$\begin{aligned}\eta, x \models \phi_1 \overset{R}{*} \phi_2 \text{ iff } \quad & x = (H, r) \text{ and exists } r_1, r_2. \text{ st.} \\ & r_1 \cdot r_2 \sqsubseteq r \\ & \text{and } \eta, (H, r_1) \models \phi_1 \text{ and } \eta, (H, r_2) \models \phi_2\end{aligned}$$

This definition looks like it might be useful to specify that we have a single data structure on the heap, but two resource views on it. A need for this kind of situation arose in the merge-sort example in Section 2.2. There, the auxiliary procedure `advance`, that advances a pointer a certain number of elements through a list, was given the specification:

$$\begin{aligned}\mathrm{Pre}(\texttt{advance}) &: \mathsf{lseg}(a_0, \texttt{l}, \mathit{null}) \\ \mathrm{Post}(\texttt{advance}) &: \mathsf{lseg}(a_0, \texttt{l}, \texttt{retval}) * \mathsf{lseg}(a_0, \texttt{retval}, \mathit{null})\end{aligned}$$

where a_0 was an amount of resource associated with every element of the list that was to be preserved. Note that `advance` does not modify the list in any way and does not consume any resources. Evidently, this specification is satisfied for any a_0. In the spirit of Separation Logic, we would like to be able to state the specification of `advance` without mentioning

resources—because it does not consume or release any—and combine the specification later on with the fact that the list has some associated resources. So, we would like to give `advance` the specification:

$$\begin{array}{rcl} \text{Pre}(\texttt{advance}) & : & \mathsf{lseg}(0, \texttt{l}, \mathit{null}) \\ \text{Post}(\texttt{advance}) & : & \mathsf{lseg}(0, \texttt{l}, \texttt{retval}) * \mathsf{lseg}(0, \texttt{retval}, \mathit{null}) \end{array}$$

since it does not require or yield any consumable resource, and then apply a putative resource-frame rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{P \overset{R}{*} R\}\ C\ \{Q \overset{R}{*} R\}}$$

Where R would record that every element of the unmodified list has a_0 associated resource. As with the normal frame rule of Separation Logic, this turns what would be a second-order universally quantified assertion into an unquantified assertion. This simplification is crucial for developing automated procedures for discharging verification conditions as in Section 5. For the present example, we can use linear programming to infer the instantiation of a_0, but for general complex composite resources, or for examples that require polymorphic recursion, this problem could become much harder.

Unfortunately, in order for this rule to be sound we must ensure that C does not modify the heap in any way that would violate the resource associations. There is currently no way to enforce this within the logic. In the example, this manifests itself as the inability to guarantee that the list in the post-condition of `advance` has exactly the same shape as the list in the precondition, which would be required to assign a resource to every element.

5. Automated Verification

In this section we describe a verification condition (VC) generation and proof search procedure for automated verification of programs against specifications in the program logic, as long as procedures have been annotated with loop invariants. The restricted subset of Separation Logic that we use in this section is similar to the subset used by Berdine et al. [6], though instead of performing a forwards analysis of the program, we generate verification conditions by backwards analysis and then attempt to solve them using proof search. We develop our own proof search procedure rather than re-use an existing Separation Logic-inspired tool in order to incorporate a key feature of Hofmann and Jost's amortised system: the use of linear programming to infer resource annotations [15]. In the system we present here, the proof search procedure generates linear constraints that can be solved by linear programming to infer resource annotations. A limitation of the VC-generate and solve technique we use here is the potential for exponential blow-up of the verification conditions in the size of program. The forward symbolic execution approach deals with this by attempting to prune unreachable paths as soon as possible and merging similar feasible paths using heuristics.

5.1. Restricted Assertion Logic.

Following Berdine et al., we make use of a highly restricted assertion logic that forbids arbitrary nesting of the additive and separating conjunctions and disallows negative occurrences of implications. This makes proof search practical. For the purposes of resource annotation inference, consumable resources are represented in

the restricted syntax as linear expressions over a collection of globally existentially quantified meta-variables. We use y_1, y_2, and so on for resource variables to be inferred, and x_1, x_2, etc. for logical variables. Resource variables cannot be quantified over inside formulae.

The basic assertion of the proof search logic is of the form

$$\bigvee_i \exists \overline{x}.\ \Pi_i \mid \Sigma_i \mid \Theta_i$$

where we use the meta-variable S to stand for such assertions. They consist of a finite disjunction of clauses, each with a collection of existentially quantified variables and three collections of assertions. The first portion, Π, contains assertions about pure (non-heap and non-consumable resource) data, which are equalities and disequalities of the form:

$$P ::= t_1 = t_2 \mid t_1 \neq t_2$$

The terms that we allow in the data and heap assertions are either variables, or the constant *null*. A collection $\Pi = P_1, ..., P_n$ is interpreted by translation into the logic of Section 4 as the additive conjunction of the P_i. The second portion, Σ, contains assertions about the heap, which are of the form:

$$X ::= [t_1 \overset{\mathsf{f}}{\mapsto} t_2] \mid \mathsf{lseg}(\Theta, t_1, t_2)$$

Here we have made use of the inductively defined list segment predicate from Section 4.2. A collection $\Sigma = X_1, ..., X_n$ is interpreted as the separating (or multiplicative) conjunction of the X_i. The final portion Θ is a linear expression indicating an amount of consumable resource. It is easily possible to generalise this to multiple resources by considering multiple named linear expressions. Given a valuation of the resource meta-variables y_i, extended to an interpretation $[\![\Theta]\!]$ this is interpreted in the logic of Section 4 as the formula $R^{[\![\Theta]\!]r}$ for some fixed resource r. A whole composite $\Pi \mid \Sigma \mid \Theta$ is interpreted as $\Pi \wedge (\Sigma * R^{[\![\Theta]\!]r})$.

Finally, we have the set of goal formulae that the verification condition generator will produce and the proof search will solve.

$$G \quad ::= \quad S * G \mid S \mathrel{-\!\!*} G \mid S \mid G_1 \wedge G_2 \mid P \to G \mid \forall x.G \mid \exists x.G$$

Note that we only allow implications ($\to$ and $\mathrel{-\!\!*}$) to appear in positive positions. This means that we can interpret them in our proof search as adding extra information to the context.

5.2. **Verification Condition Generation.** Verification condition generation is performed for each procedure individually by computing weakest liberal preconditions for each instruction, working backwards from the last instruction in the method. To resolve loops, we require that the targets of all backwards jumps have been annotated with loop invariants S that are of the special form in the previous section. This assumes that the instructions have been sorted into reverse post-order so we can scan the instructions in reverse order to collect the verification conditions. We omit the rules that we use for weakest liberal precondition generation since they are very similar to the rules for the shallowly embedded logic in Figure 5. The verification condition generator will always produce a VC for the entailment of the computed weakest liberal precondition of the first instruction from the procedure's precondition, plus a VC for each annotated instruction, being the entailment between the annotation and the computed weakest liberal precondition. All VCs will have a formula of the form S as the antecedent and a goal formula G as the conclusion.

The verification condition generation procedure has been formalised within the Coq proof assistant and proved sound with respect to the program logic in Section 3. By using Coq's module system [12] we have abstracted the verification condition generator over the particular deep assertion logic used. We used Coq's program extraction capabilities to extract the verification condition generator and instantiated it with the proof search logic described in this section.

5.3. **Proof Search.** The output of the verification condition generation phase is a collection of problems of the form $S \vdash G$, which can each be reduced to a finite collection of sequents of the form $\Pi \mid \Sigma \mid \Theta \vdash G$. To discharge these proof obligations, we make use of the I/O interpretation of proof search as defined for intuitionistic linear logic by Cervesato, Hodas and Pfenning [9], along with heuristic rules for unfolding the inductive list segment predicate. We augment the I/O model of resource accounting with an additional part that collects linear constraints that may be fed into a integer linear program solver, to automatically infer resource annotations.

We use the following judgement form for proof search goals that collect linear constraints. Here $\mathcal{C}$ is a set of linear constraints over the resource meta-variables y_i:

$$\Pi \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}$$

The proof search procedure is defined by the rules shown in Figure 8, Figure 9, Figure 10 and Figure 11. These rules make use of several auxiliary judgements:

$\Pi \mid \Sigma \mid \Theta \vdash \Sigma_{goal} \setminus \Sigma_{out}, \Theta_{out}, \mathcal{C}$	Heap assertion matching
$\Theta \vdash \Theta_{goal} \setminus \Theta_{out}, \mathcal{C}$	Resource matching
$\Pi \vdash \bot$	Contradiction spotting
$\Pi \vdash \Pi'$	Data assertion entailment

The backslash notation used in these judgements follows Cervesato et al., where in the judgement $\Theta \vdash \Theta_{goal} \setminus \Theta_{out}, \mathcal{C}$, the proof context Θ denotes the facts used as input and Θ_{out} denotes the facts that are left over (the output) from proving Θ_{goal}. The $\mathcal{C}$ component collects the linear constraints that must hold for the judgement to give a valid separation logic entailment. A similar interpretation is used for the heap assertion matching judgement. We do not define the data entailment or contradiction spotting judgement explicitly here; we intend that these judgements satisfy the basic axioms of equalities and disequalities.

The rules in Figure 8 are the goal driven search rules. There is an individual rule for each possible kind of goal formula. The first two rules are matching rules that match a formula S against the context, altering the context to remove the heap and resource assertions that S requires, as dictated by the semantics of the assertion logic. We must search for a disjunct i that is satisfied by the current context. There may be multiple such i, and in this case the search may have to backtrack. When the goal is a formula S, then we ask that the left-over heap is empty, in order to detect memory leaks. Note that the logical variables $x_1, x_2, ...$ may not occur in the constraint sets, so we do not need to handle universally quantified constraints.

The matching rules make use of the heap and resource matching judgements defined in Figure 9. The heap matching judgements take a data, heap and resource context and attempt to match a list of heap assertions against them, returning the left over heap, resources and computed constraints. The first three rules are straightforward: the empty heap assertion is always matchable, points-to relations are looked up in the context directly and pairs of heap assertions are split, threading the contexts through. For the list segment

$$\frac{\begin{array}{c}\text{exists } i, \overline{t}. \qquad \Pi \mid \Sigma \mid \Theta \vdash \Sigma_i[\overline{t}/\overline{x}] \setminus \Sigma', \Theta', \mathcal{C}_H \\ \Pi \vdash \Pi_i[\overline{t}/\overline{x}] \qquad \Theta' \vdash \Theta_i[\overline{t}/\overline{x}] \setminus \Theta'', \mathcal{C}_R \qquad \Pi \mid \Sigma' \mid \Theta'' \vdash G \setminus \mathcal{C}_G\end{array}}{\Pi \mid \Sigma \mid \Theta \vdash \left(\bigvee_i \exists \overline{x}.\ \Pi_i \mid \Sigma_i \mid \Theta_i\right) * G \setminus \mathcal{C}_H \cup \mathcal{C}_R \cup \mathcal{C}_G}$$

$$\frac{\text{exists } i, \overline{t}. \qquad \Pi \mid \Sigma \mid \Theta \vdash \Sigma_i[\overline{t}/\overline{x}] \setminus \mathsf{emp}, \Theta', \mathcal{C}_H \qquad \Pi \vdash \Pi_i \qquad \Theta' \vdash \Theta_i \setminus \Theta'', \mathcal{C}_R}{\Pi \mid \Sigma \mid \Theta \vdash \bigvee_i \exists \overline{x}.\ \Pi_i \mid \Sigma_i \mid \Theta_i \setminus \mathcal{C}_H \cup \mathcal{C}_R}$$

$$\frac{\text{forall } i, \overline{x}. \qquad \Pi, \Pi_i \mid \Sigma, \Sigma_i \mid \Theta + \Theta_i \vdash G \setminus \mathcal{C}}{\Pi \mid \Sigma \mid \Theta \vdash \left(\bigvee_i \exists \overline{x}.\ \Pi_i \mid \Sigma_i \mid \Theta_i\right) \mathrel{-\!\!*} G \setminus \mathcal{C}} \qquad \frac{\Pi, P \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}}{\Pi \mid \Sigma \mid \Theta \vdash P \to G \setminus \mathcal{C}}$$

$$\frac{\Pi \mid \Sigma \mid \Theta \vdash G_1 \setminus \mathcal{C}_1 \qquad \Pi \mid \Sigma \mid \Theta \vdash G_2 \setminus \mathcal{C}_2}{\Pi \mid \Sigma \mid \Theta \vdash G_1 \wedge G_2 \setminus \mathcal{C}_1 \cup \mathcal{C}_2} \qquad \frac{\Pi \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C} \qquad x \notin fv(\Pi) \cup fv(\Sigma)}{\Pi \mid \Sigma \mid \Theta \vdash \forall x.G \setminus \mathcal{C}}$$

$$\frac{\Pi \mid \Sigma \mid \Theta \vdash G[t/x] \setminus \mathcal{C}}{\Pi \mid \Sigma \mid \Theta \vdash \exists x.G \setminus \mathcal{C}}$$

Figure 8: Goal Driven Search Rules

rules, there are three cases. Either the two pointers involved in the list are equal, in which case we are immediately done; or we have a single list cell in the context that matches the start pointer of the predicate we are trying to satisfy, and we have the required resources for an element of this list, so we can reduce the goal by one step; or we have a whole list segment in the context and we can reduce the goal accordingly. The resource matching rule is where linear constraints are actually generated; to match a resource we subtract the desired resource from the available resources and add a constraint to ensure that there was enough resource to do this. Note that this rule always succeeds during proof search, but may generate unsatisfiable constraints. Thus back-tracking may still be required.

The final two sets of rules operate on the proof search context. The first set, shown in Figure 10, describe how information flows from the heap part of the context to the data part. If we know that two variables both have a points-to relation involving a field f, then we know that these locations must not be equal. Similarly, if we know that a variable does point to something, then it cannot be null. If any contradictions are found using these rules, then the proof search can terminate immediately for the current goal. This is provided for by the first rule in Figure 10.

The final set of rules performs heuristic unfolding of the inductive lseg predicate. These rules are shown in Figure 11. These rules take information from the data context and use it to unfold lseg predicates that occur in the heap context. The first rule is triggered when the proof search learns that there is a list segment where the head pointer of the list is not equal to null. In this case, two proof search goals are produced, one for the case that

Heap Matching Rules:

$$\frac{}{\Pi \mid \Sigma \mid \Theta \vdash \mathsf{emp} \setminus \Sigma, \Theta, \{\}} \qquad \frac{\Pi \vdash t_1 = t_1' \qquad \Pi \vdash t_2 = t_2'}{\Pi \mid \Sigma, [t_1 \overset{\mathsf{f}}{\mapsto} t_2] \mid \Theta \vdash [t_1' \overset{\mathsf{f}}{\mapsto} t_2'] \setminus \Sigma, \Theta, \{\}}$$

$$\frac{\Pi \mid \Sigma \mid \Theta \vdash \Sigma_1 \setminus \Sigma', \Theta', \mathcal{C}_1 \qquad \Pi \mid \Sigma' \mid \Theta' \vdash \Sigma_2 \setminus \Sigma'', \Theta'', \mathcal{C}_2}{\Pi \mid \Sigma \mid \Theta \vdash \Sigma_1 * \Sigma_2 \setminus \Sigma'', \Theta'', \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{\Pi \vdash t_1 = t_2}{\Pi \mid \Sigma \mid \Theta \vdash \mathsf{lseg}(\Theta_l, t_1, t_2) \setminus \Sigma, \Theta, \{\}}$$

$$\frac{\Pi \vdash t_1 = t_1' \qquad \Theta \vdash \Theta_l \setminus \Theta', \mathcal{C}_R \qquad \Pi \mid \Sigma \mid \Theta' \vdash \mathsf{lseg}(\Theta_l, t_n, t_2) \setminus \Sigma', \Theta'', \mathcal{C}_H}{\Pi \mid \Sigma, [t_1 \overset{\mathsf{next}}{\mapsto} t_n], [t_1 \overset{\mathsf{data}}{\mapsto} t_d] \mid \Theta \vdash \mathsf{lseg}(\Theta_l, t_1', t_2) \setminus \Sigma', \Theta'', \mathcal{C}_R \cup \mathcal{C}_H}$$

$$\frac{\Pi \vdash t_1' = t_1 \qquad \Pi \mid \Sigma \mid \Theta \vdash \mathsf{lseg}(\Theta_l, t_2, t_3) \setminus \Sigma', \Theta', \mathcal{C}}{\Pi \mid \Sigma, \mathsf{lseg}(\Theta_l, t_1, t_2) \mid \Theta \vdash \mathsf{lseg}(\Theta_l, t_1', t_3) \setminus \Sigma', \Theta', \mathcal{C}}$$

Resource Matching Rule:

$$\frac{}{\Theta \vdash \Theta' \setminus (\Theta - \Theta'), \{\Theta \geqslant \Theta'\}}$$

Figure 9: Matching Rules

$$\frac{\Pi \vdash \bot}{\Pi \mid \Sigma \mid \Theta \vdash G \setminus \{\}}$$

$$\frac{\Sigma = [t_1 \overset{\mathsf{f}}{\mapsto} t], [t_2 \overset{\mathsf{f}}{\mapsto} t'], \Sigma' \qquad \Pi, t_1 \neq t_2 \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}}{\Pi \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}}$$

$$\frac{\Sigma = [t \overset{\mathsf{f}}{\mapsto} t'], \Sigma' \qquad \Pi, t \neq \mathit{null} \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}}{\Pi \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}}$$

Figure 10: Contradiction Flushing

the list segment is empty and one for when it is not. The other rules are similar; taking information from the data context and using it to refine the heap context.

The proof search strategy that we employ works by first saturating the context by repeatedly applying the rules in Figure 10 and Figure 11 to move information from the data context into the heap context and vice versa. This process terminates because there are a finite number of points-to relations and list segment predicates to generate rule applications, and when new predicates are introduced via list segment unfolding they either do not trigger any new inequalities or are over fresh variables about which nothing is yet known. Once

$$\frac{\Pi \vdash t_1 \neq \mathit{null} \qquad \Pi, t_1 = t_2 \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}_1 \qquad \Pi \mid \Sigma, [t_1 \overset{\mathsf{next}}{\mapsto} x], [t_1 \overset{\mathsf{data}}{\mapsto} y], \mathsf{lseg}(R, x, t_2) \mid \Theta, R \vdash G \setminus \mathcal{C}_2}{\Pi \mid \Sigma, \mathsf{lseg}(R, t_1, t_2) \mid \Theta \vdash G \setminus \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{\Pi \vdash t_1 = \mathit{null} \qquad \Pi, t_2 = \mathit{null} \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}}{\Pi \mid \Sigma, \mathsf{lseg}(R, t_1, t_2) \mid \Theta \vdash G \setminus \mathcal{C}} \qquad \frac{\Pi \vdash t_1 = t_2 \qquad \Pi \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}}{\Pi \mid \Sigma, \mathsf{lseg}(R, t_1, t_2) \mid \Theta \vdash G \setminus \mathcal{C}}$$

$$\frac{\Pi \vdash t_1 \neq t_2 \qquad \Pi \mid \Sigma, [t_1 \overset{\mathsf{next}}{\mapsto} x], [t_1 \overset{\mathsf{data}}{\mapsto} y], \mathsf{lseg}(R, x, t_2) \mid \Theta, R \vdash G \setminus \mathcal{C}}{\Pi \mid \Sigma, \mathsf{lseg}(R, t_1, t_2) \mid \Theta \vdash G \setminus \mathcal{C}}$$

Figure 11: List Unfolding Rules

the context is fully saturated, the proof search reduces the goal by using the goal-driven search rules and the process begins again.

Given a collection of verification conditions and a successful proof search over them that has generated a set of linear constraints, we input these into a linear solver, along with the constraint that every variable is positive and an objective function that attempts to minimise variables appearing in the precondition.

Theorem 5.1. *The proof search procedure is terminating. Moreover, it is sound: if* $\Pi \mid \Sigma \mid \Theta \vdash G \setminus \mathcal{C}$ *and there is an valuation of the* $\overline{y}$ *that satisfies* $\mathcal{C}$*, then* $\Pi \wedge (\Sigma * R^{[\![\Theta]\!]_r}) \vdash G$ *under this valuation using the translation of the proof search logic into the logic of Section 4 as defined in Section 5.1.*

6. Examples of Automated Verification

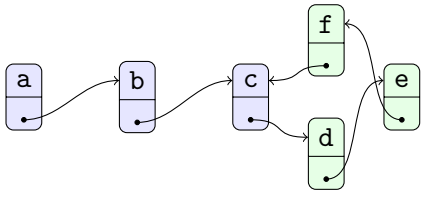
6.1. **Small Examples.** We have tested the automated resource inference procedure described in the previous section on several small examples, including the simple loop iteration example from the introduction and the examples from Section 2. We summarise these examples in the following table. Note that the proof search procedure also verifies the memory safety of these examples. For timings, these tests were performed on an PC running 64-bit Ubuntu Linux 10.10 on an 8-core Intel Core i7 860 at 2.80GHz. Times were measured using the GNU `time` utility. Our implementation only makes use of a single core.

Name	Property Inferred	Time (s)
`iterate_list`	Number of iterations is length of input list	0.010
`iterate_recursive`	Number of calls is length of input list	0.010
`copy_list`	Number of allocations is length of input list	0.013
`reverse`	Number of iterations is length of input list	0.012
`queue`	Resource annotations on `enqueue` and `deqeueue`	0.019
`frying_pan`	Number of times each element is visited	0.018
`mergesort`	Number of comparisons is length of input list	0.037
`tree_traverse`	Number of calls is size of input tree	0.012
`tree_copy`	Number of allocations is size of input tree	0.014
`tree_mirror`	Number of calls is size of input tree	0.012

In each of these examples, we seeded the program with loop invariants describing the shape of heap data structures, but left our implementation to infer the resource bounds. The last three examples involving trees make use of the inductive tree predicate from Section 4.2. The proof search procedure was augmented with heuristic rules for matching and unfolding instances of the tree predicate, following those for lists. The tree examples are all recursive procedures.

As can be seen from the table, the time taken for each of the examples is trivial. It remains to be seen how well this technique scales to real-world code. It seems evident that the VC-generate and solve process is not scalable in general due to the potential for exponential blow-up in the size of the generated formulae. A more realistic implementation of the program logic described in this paper would likely use the forward symbolic execution approach, as described by Berdine et al. [6].

6.2. **Frying Pan List Reversal.** As a larger example, we demonstrate the use of the proof search procedure coupled with linear constraint generation on the standard imperative in-place list reversal algorithm on lists with cyclic tails (also known as "frying pan" lists). This example was used by Brotherston, Bornat and Calcagno [7] to illustrate the use of cyclic proofs to prove program termination. Here we show how our amortised resource logic can be used to infer bounds on the time complexity of this procedure.



The "handle" of the structure consists of the nodes `a`, `b`, `c` and the "pan" consists of the nodes `d`, `e` and `f`. When the in-place list-reversal procedure is run upon a structure of this shape, it will proceed up the handle, reversing it, around the pan, reversing it, and then back down the handle, restoring it to its original order. For the purposes of this example, we assume that it takes one element of resource to handle the reversal of one node. Following Brotherston, Bornat and Calcagno, we can specify a cyclic list in Separation Logic by the following formula, where v_0 points to the head of the list and v_1 points to the join between

the handle and the pan[4].

$$\exists k.\mathsf{lseg}(x_1, v_0, v_1) * [v_1 \overset{\mathsf{next}}{\mapsto} k] * \mathsf{lseg}(x_2, k, v_1) * R^{x_3}$$

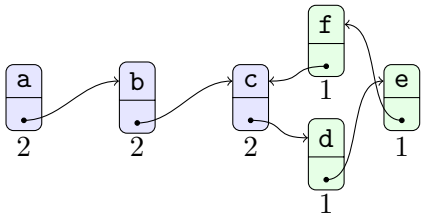
We have annotated the list segments involved with resource annotation variables x_1 and x_2 that we will instantiate using linear programming. The predicate R^{x_3} denotes any extra resource we may require. Similarly, we have annotated the required loop invariant (adapted from Brotherston et al.):

$$\begin{array}{l}(\exists k.\ \mathsf{lseg}(a_1, l_0, v_1) * \mathsf{lseg}(a_2, l_1, \mathit{null}) * [v_1 \overset{\mathsf{next}}{\mapsto} k] * \mathsf{lseg}(a_3, k, v_1) * R^{a_4}) \\ \vee\ (\exists k.\ \mathsf{lseg}(b_1, k, \mathit{null}) * [j \overset{\mathsf{next}}{\mapsto} k] * \mathsf{lseg}(b_2, l_0, v_1) * \mathsf{lseg}(b_3, l_1, j) * R^{b_4}) \\ \vee\ (\exists k.\ \mathsf{lseg}(c_1, l_0, \mathit{null}) * \mathsf{lseg}(c_2, l_1, v_1) * [v_1 \overset{\mathsf{next}}{\mapsto} k] * \mathsf{lseg}(c_3, k, v_1) * R^{c_4})\end{array}$$

Each disjunct of the loop invariant corresponds to a different phase of the procedure's progress. Brotherston et al. note that it is possible to infer the shape part of this loop invariant using current Separation Logic tools. Here, we have added the ability to infer resource annotations, and hence bounds on the time consumption of the procedure. Running our tool on this example produces the following instantiation of the variables:

Precondition	$x_1 = 2$	$x_2 = 1$	$x_3 = 2$	
Loop invariant, phase 1	$a_1 = 2$	$a_2 = 1$	$a_3 = 1$	$a_4 = 2$
Loop invariant, phase 2	$b_1 = 1$	$b_2 = 1$	$b_3 = 0$	$b_4 = 1$
Loop invariant, phase 3	$c_1 = 1$	$c_2 = 0$	$c_3 = 0$	$c_4 = 0$
Post-condition	$x'_1 = 0$	$x'_2 = 0$	$x'_3 = 0$	

Pictorially, the inference has associated the following amount of resource with each part of the input structure:



Each node of the handle has 2 associated elements of resource, to handle the two passes of the handle that the procedure takes, while the pan has one element of resource for each node. The inferred annotations for the loop invariant track how the resources on each node are consumed by the procedure, gradually all reducing to zero. Since we have added a consume instruction to be executed every time the procedure starts a loop, the resource inference process has also verified the termination of this procedure, and given us a bound on the number of times the loop will execute in terms of the shape of the input.

7. Conclusions

We have presented a program logic that extends the resource reasoning capabilities of Separation Logic from reasoning about mutable resources such as the heap to consumable resources such as time. We have demonstrated how doing so allows tight connections between the shape of data structures and the resources required to process them to be stated,

[4]Note that the data part of the list node has been omitted in this formula and in the loop invariants to reduce clutter.

and so expanding the reach of Separation Logic's local reasoning principle to consumable resources.

We have presented an automated proof procedure that takes programs annotated with shape invariants and infers consumable resource annotations. The main limitation of this automated proof search procedure is that it only supports the statement and inference of bounds that are linear in the size of lists that are mentioned in a procedure's precondition. This is a limitation shared with the original work of Hofmann and Jost [15]. We note that this is not a limitation of the program logic that we have presented, only of the automated verification procedure that we have layered on top. In Section 4.2 we presented a Separation Logic version of the polynomial potential lists of Hoffmann and Hofmann [14], which opens the way to inference of polynomial bounds for pointer manipulating list programs. Initial experiments with extending our implementation in this direction have been promising.

We have demonstrated that the use of mixed shape and resource assertions can simplify the complexity of specifications that talk about resources, and this should extend to extensions of the proof search procedure, or to interactive systems based on this program logic. The resource aware program logic of Aspinall et al. [3] also uses the same layering: a general program logic for resources (which is proved complete in their case) is used as a base for a specialised logic for reasoning about the output of the Hofmann-Jost system.

A possible direction for future work is to consider different assertion logics and their expressiveness in terms of the magnitude of resources they can express. We conjecture that the deep assertion logic we have presented here, extended with the lseg predicate can express resources linear in the size of the heap. It would be interesting to consider more expressive logics and evaluate them from the point of view of implicit computational complexity; the amount of resource that one can express in an assertion dictates the amount of resource that is available for the future execution of the program.

Additional future work is to consider the proof theory of the combined Boolean BI and affine intuitionistic BI that have used in this paper.

Other resource inference procedures that are able to deal with non-linear bounds include those of Chin et al. [10, 11], Albert et al. [1] and Gulwani et al. [13]. When dealing with heap-based data structures, all of these techniques use a method of attaching size information to assertions about data structures. As we demonstrated in Section 2.1, this can lead to unwanted additional complexity in specifications. However, all of these techniques deal with numerically bounded loops, which our current prototype automated procedure cannot. We are currently investigating how to extend our approach to deal with non-linear and numerically-driven resource bounds.

Acknowledgements I would like to thank Kenneth MacKenzie and Brian Campbell for discussion and comments on this work. The ESOP 2010 and LMCS anonymous reviewers also provided helpful suggestions. This work was funded by EPSRC Follow-on Fund grant EP/G006032/1 "Resource Static Analysis" and EPSRC grant EP/G068917/1 "Categorical Foundations for Indexed Programming".

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26,*

2007, Revised Lectures, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2007.

[2] David Aspinall, Robert Atkey, Kenneth MacKenzie, and Donald Sannella. Symbolic and Analytic Techniques for Resource Analysis of Java Bytecode. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Proceedings of 5th International Symposium on Trustworthy Global Computing (TGC 2010)*, volume 6084 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2010.

[3] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007.

[4] David Aspinall, Patrick Maier, and Ian Stark. Safety guarantees from explicit resource management. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 2007.

[5] Robert Atkey. Amortised Resource Analysis with Separation Logic. In Andrew Gordon, editor, *ESOP 2010: Proceedings of the 19th European Symposium on Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2010.

[6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[7] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 101–112. ACM, 2008.

[8] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, 1982.

[9] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theor. Comput. Sci.*, 232(1-2):133–163, 2000.

[10] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing memory resource bounds for low-level programs. In Richard Jones and Stephen M. Blackburn, editors, *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*, pages 151–160. ACM, 2008.

[11] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin C. Rinard. Memory usage verification for oo programs. In Chris Hankin and Igor Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2005.

[12] Jacek Chrzaszcz. Implementing Modules in the Coq System. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.

[13] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 127–139. ACM, 2009.

[14] Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In Andrew Gordon, editor, *ESOP 2010: Proceedings of the 19th European Symposium on Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[15] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '03, pages 185–197, New York, NY, USA, 2003. ACM.

[16] Martin Hofmann and Steffen Jost. Type-Based Amortised Heap-Space Analysis. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[17] Martin Hofmann and Dulma Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009.

[18] Robert Hood and Robert Melville. Real-Time Queue Operations in Pure LISP. *Information Processing Letters*, 13(2):50–53, 1981.

[19] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM.

[20] P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–243, 1999.

[21] Chris Okasaki. *Purely Functional Data Stuctures*. Cambridge University Press, 1998.

[22] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[23] Greg Restall. *An Introduction to Substructural Logics*. Routledge, 2000.

[24] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[25] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.